16 | 怎么在遗留系统上写测试？

2021-09-08 郑晔

《程序员的测试课》 课程介绍 >

讲述：郑晔
时长 11:39 大小 10.67M

你好，我是郑晔！

迄今为止，我们讨论的话题主要是围绕着如何在一个新项目上写测试。但在真实世界中，很多人更有可能面对的是一个问题重重的遗留系统。相比于新项目，在一个遗留系统上，无论是写代码还是写测试，都是一件有难度的事。

在讨论如何在遗留系统上写测试前，我们首先要弄清楚一件事：什么样的系统算是遗留系统。在各种遗留系统的定义中，Michael Feathers 在《修改代码的艺术》（Working Effectively with Legacy Code）中给出的定义让我印象最为深刻——遗留系统就是没有测试的系统。

根据这个定义你会发现，即便是新写出来的系统，因为没有测试，它就是遗留系统。由此可见测试同遗留系统之间关系之密切。想要让一个遗留系统转变成为一个正常的系统，关键点就是写测试。

给遗留系统写测试

众所周知，给遗留系统写测试是一件很困难的事情。但你有没有想过，为什么给遗留系统写测试很困难呢？

如果代码都写得设计合理、结构清晰，即便是补测试也困难不到哪去。但大部分情况下，我们面对的遗留系统都是代码冗长、耦合紧密。你会不会一想到给遗留系统写测试就头皮发麻？因为实在是太麻烦了。由此我们知道，给遗留系统写测试，难点不在于测试，而在于它的代码。

如果不能了解到一个系统应该长成什么样子，我们即便努力做到了局部的一些改进，系统也会很快地退化成原来的样子。这也是为什么我们学习写测试要从一个新项目开始，因为我希望你对一个正常系统的样子有个认知，**写测试不只是写测试的事，更是写代码的事**。

在遗留系统上写测试，本质上就是一个系统走向正常的过程。对于一个系统来说，一旦能够正常运行，最好的办法就是不动它，即便是要给它写测试。但在真实世界中，一个有生命力的系统总会有一些让我们不得不去动它的理由，可能是增加新特性，可能是要修改一个 Bug，也可能是要做系统的优化。

我们不会一上来就给系统完整地添加测试，这也几乎是不可能完成的任务。所以，本着实用的态度，我们的做法是，**动到哪里，给哪里写测试**。

要动哪里，很大程度上就是取决于我们对既有代码库的理解。不过，既然是遗留代码，可能出现的问题是，你不一定理解你要修改的这段代码究竟是怎么起作用的。最有效的解决办法当然是找懂这段代码的人请教一番，但如果你的代码库生命周期够长，很有可能已经没有人知道这段代码是怎么来的了。在如今这个时代里，摸黑看代码时，我们可以使用 IDE 提供的重构能力，比如提取方法，将大方法进行拆分，这样有助于降低难度。

至于给哪里写测试，最直观的做法当然是编写最外层的系统测试。这种做法可行，但正如我们在上一讲所说，越是外层的测试，编写的成本越高，执行速度越慢。虽然覆盖面会广一些，但具体到我们这里要修改代码而言，存在一种可能就是控制得不够准确。换言之，

很有可能我们写了一个测试，但是我们改不改代码，对这个测试影响不大。所以，只要有可能，我们还是要努力地降低测试的层次，更精准地写测试。也就是能写集成测试，就不写系统测试；能写单元测试，就不写集成测试。

或许你会说，我也知道能写单元测试很好，但通常遗留系统最大的问题就在于单元测试不好写。**造成测试不好写的难点就是耦合**，无论是代码与外部系统之间的耦合，还是代码与第三方程序库的耦合，抑或是因为代码写得不好，自己的代码就揉成了一团。所以，**想在遗留系统中写好测试，一个关键点就是解耦**。

一个解耦的例子

我们在专栏前面中讲过，测试的关键就在于构建一个可控的环境。对于编写单元测试来说，可控环境很关键的一步就是使用模拟对象，也就是基于 Mock 框架生成的对象。

同样，在遗留系统上如果想要编写单元测试，模拟对象也很关键。换言之，我们要给一个类编写单元测试，首先要把它周边的组件由模拟对象替换掉，让它有一个可控的环境。说起来很简单，但面对遗留系统时，想要用模拟对象替换掉真实对象就不是一件轻松的事。

下面我们就用一个例子看看如何在一个遗留系统上进行解耦，然后又是如何给代码写测试。我们有一个订单服务，完成了下单过程之后，要发出一个通知消息给到 Kafka，以便通知下游的服务。

复制代码

```
1  public class OrderService {
2    private KafkaProducer producer;
3
4    public void placeOrder(final OrderParameter parameter) {
5      ...
6      this.producer.send(
7        new ProducerRecord<String, String>("order-topic", DEFAULT_PARTITION, Int
8      );
9    }
10 }
```

很显然，这段代码我们直接依赖了 KafkaProducer，这是 Kafka 提供的 API，如果要想测试 OrderService 这个类，我们就需要把 Kafka 加到这个测试里，而我们的测试重点是下

单的过程，这个过程本身同 Kafka 没有关系。要测试这个类，我们必须把 Kafka 从我们的代码中解耦开。

首先，我们用**提取方法（Extract Method）**这个重构手法把 Kafka 相关的代码调用封装起来，通过使用 IDE 的重构功能就可以完成。

复制代码

```
public class OrderService {
  private KafkaProducer producer;

  public void placeOrder(final OrderParameter parameter) {
    ...
    send(orderId);
  }

  private void send(final OrderId orderId) {
    this.producer.send(
      new ProducerRecord<String, String>("order-topic", DEFAULT_PARTITION, Int
    );
  }
}
```

接下来，我们要把 KafkaProducer 与我们的业务代码分离开。正如我们在之前讨论的内容所说，我们需要有一个封装层，把对第三方程序库的访问封装进去。所以，我们在这里引入一个新的类承担这个封装层的作用。我们可以使用 ** 提取委托（Extract Delegate）**创建出一个新的类，提取的时候，我们还要选上**生成访问器（Generate Accessors）** 的选项，它会为我们生成对应的 Getter。

复制代码

```
public class KafkaSender {
  private KafkaProducer producer;

  public KafkaProducer getProducer() {
        return producer;
  }

  ...
}
```

而 OrderService 的 send 方法就变成了下面的样子。

复制代码

```
class OrderService {
  ...
  private void send(final OrderId orderId) {
    this.kafkaSender.getProducer().send(
            new ProducerRecord<String, String>("order-topic", DEFAULT_PARTITIO
    );
  }
}
```

很显然，从当前的实现看，它只与 KafkaSender 相关，接下来，我们可以使用**搬移实例方法（Move Instance Method）**把它搬移到 KafkaSender 中。

复制代码

```
class KafkaSender {
  ...

  public void send(final OrderId orderId, OrderService orderService) {
    getProducer().send(
            new ProducerRecord<String, String>("order-topic", DEFAULT_PARTITIO
    );
  }
}

class OrderService {
  ...

  public void placeOrder(final OrderParameter parameter) {
    ...
    kafkaSender.send(orderId, this);
  }
}
```

从代码上我们可以看到，虽然 KafkaSender 的 send 方法有 OrderService 这个参数，但是我们并没有用它，可以安全地删除它（Safe Delete），这也是一个快捷键就可以完成的工作。还有，这里用到 getProducer 方法，因为我们在 KafkaSender 这个类里面了，所以，我们就不需要通过 Getter 访问了，可以通过**内联方法（Inline Method）**将它去掉。

复制代码

```
class KafkaSender {
  ...

  public void send(final OrderId orderId) {
```

```
    producer.send(
            new ProducerRecord<String, String>("order-topic", DEFAULT_PARTITIO
    );
  }
}

class OrderService {
  ...

  public void placeOrder(final OrderParameter parameter) {
    ...
    kafkaSender.send(orderId);
  }
}
```

到这里，我们的业务代码（OrderService）已经不再依赖于 KafkaProducer 这个第三方的代码，而是依赖于我们自己的封装层，这已经是一个进步了。不过从软件设计上讲，KafkaSender 是一个具体的实现，它不应该出现在业务代码中。所以，我们还需要再进一步，提取出一个接口，让我们的业务类不依赖于具体的实现。回到代码上，我们可以在 KafkaSender 这个类上执行**提取接口（Extract Interface）**这个重构动作，创建出一个新的接口。

复制代码

```
public interface Sender {
  void send(OrderId orderId);
}

public class KafkaSender implements Sender {
  @Override
  public void send(final OrderId orderId) {
    producer.send(
            new ProducerRecord<String, String>("order-topic", DEFAULT_PARTITIO
    );
  }
}

public class OrderService {
  private final Sender sender;

  public OrderService(Sender sender) {
    this.sender = sender;
  }

  ...
}
```

经过这番改造，OrderService 这个业务类已经与具体的实现完全无关了。我们就可以用模拟对象模拟出 sender，用完全可控的方式给这个类添加测试了。

复制代码

```
class OrderServiceTest {
  private OrderService service;
  private Sender sender;

  @BeforeEach
  public void setUp() {
    this.sender = mock(Sender.class);
    this.service = new OrderService(this.sender);
  }
  ...
}
```

到这里，你或许会有一个疑问，我在这里改动了这么多的代码，真的没问题吗？如果这些代码是我们手工修改，这确实是个问题。不过，现在借助 IDE 的重构功能，我们并没有手工修改任何代码，相比于过去，这也是今天做遗留系统调整的优势所在。由此可见，**理解重构，尤其是借助 IDE 的重构功能，是我们更好地去做遗留系统调整的基础。**否则，我们必须先构建更外层的测试，无论是系统测试还是人工测试。

现在我们来回顾一下前面做了些什么。首先，我们有一个大目标：为了能够有效地测试，我们需要把具体实现和业务解耦开。在前面的例子中，主要就是要把 KafkaProducer 从业务类中分开。

把具体实现的代码从业务实现中隔离开，我们采用的手法是提取方法，这一步是为了后面把具体实现从业务类中挪出去做准备。通过引入一个封装类（KafkaSender），我们将具体的实现（KafkaProducer）从业务类中挪了出去。

到这里，我们的业务类已经完全依赖自己编写的代码。不过，这个封装类还是特定于具体的实现，让业务依赖于一个具体实现在设计上也是不恰当的。所以，我们这里再进一步，提取出一个接口。

从软件设计的角度看，这个提取出来的接口就是这个设计中缺失的一个模型，所以，提取这个接口不是画蛇添足，而恰恰是补齐了之前在设计上的欠缺。

换个角度看，模拟对象模拟的是接口行为，而很多遗留代码只有具体的类，而没有接口。虽然有些具体类也是可以模拟的，但出于统一原则的考虑，我们应该**针对所有具体类提取一个接口出来**，而让原来的类成为实现这个接口的一个实现类。有了接口，我们也就可以使用模拟对象，做行为可控的测试了。

这一系列的做法非常有用，比如，业务代码中调用了 static 方法，它在测试中也不好模拟。我们也可以通过提取方法把它隔离出来，然后把它挪到一个封装类里面，引入一个新的接口，让一段无法模拟的代码变得可以模拟。如果你真的能够理解这种做法，已经可以消灭掉很多设计不好的代码了。

当然，这里没有涵盖在遗留系统上写测试的各种做法，但你已经掌握了最精髓的部分：**先隔离，再分离**。如果你有兴趣了解更多的做法，推荐一本书给你，就是前面提到的《修改代码的艺术》（Working Effectively with Legacy Code）。虽然它是一本介绍处理遗留代码的书，在我看来，它更是一本教人如何写测试的书。

总结时刻

今天我们谈到了在遗留系统上写测试。遗留系统就是那些没有测试的系统，给遗留系统写测试就是让一个系统恢复正常的过程。

在遗留系统上做改进，关键是要知道改进成什么样子。在一个遗留系统上写测试，不仅是写测试，还会牵扯到写代码。

完整地给一个遗留系统写测试是比较困难的。一个实用的改进策略是，动到哪里，改哪里。具体如何写测试，最好是测试的层次越低越好，但低层次的测试就会涉及代码耦合的问题，而这里就需要我们对代码进行解耦。

解耦，主要是把业务代码和具体实现分开。通过提取方法，把一段耦合紧密的代码隔离开，再创建一个新的封装类把它挪进去。如果代码里有很多具体类，我们还可以通过引入接口进行解耦。这里面的关键是利用 IDE 给我们提供的重构功能，减少手工改代码的操作。

如果今天的内容你只能记住一件事，那请记住：**改造遗留系统的关键是解耦**。

思考题

你有遗留系统改造的经验吗？你是怎么保证改造的正确性的呢？欢迎在留言区分享你的经验。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

赞 3 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 15 | 测试应该怎么配比？

下一篇 17 | TDD 就是先写测试后写代码吗？

精选留言 (6)

写留言

亦无
2021-09-16

提取独立接口确实是个好办法，和这个类似的，还有提取同类项的做法，就是同一份代码被不同地方进行了拷贝，也是可以改为统一调用，让逻辑更清晰。

现实中对于老代码的维护，大部分碰到的困境是，因为需求，需要修改老代码，修改之后没法确认影响范围，就算做了单元测试，系统测试层面为了保险起见，也是要进行大面...

展开

1

阿姆斯壮
2021-09-08

工作大多数场景是界面编程。想咨询一下校长，自己总感觉界面这块有点无法套进单元测试里面。也不知道那里欠缺了。

展开

作者回复: 现在的前端都有特定的框架支持了，比如，Selenium/Webdriver 测试前端界面，支持模拟界面上的操作，移动端也有对应的框架。不过，一般来说，前端一定要把逻辑和前端界面分开。这种框架对前端界面的支持，属于集成测试的范畴，而单元测试主要以测业务逻辑为主。

1 1

大碗

2021-09-08

如果新加一个订单完成的业务，需要发布带订单orderId,finishedTime的订单完成事件，也是在sender里面加这个方法么？怎么设计会更好呢

展开

作者回复: 一种做法是定义一个模型叫事件，不同的地方发出不同的事件，至于 orderId、finishedTime，就是事件的参数。这样，sender 有一个接口就够了。

1

闻人

2021-09-08

要让项目易于测试，写代码要注重隔离，实现与接口隔离，业务与外部组件隔离 #收纳盒 #极客时间

作者回复: 其实就是做好设计

1

刘大明

2021-09-17

郑大，有个问题请教一下，如果每次重构的时候发现需要提取一些类，怎么将这些单独提取出来的方法放在合适的位子呢。比方说有一块重复代码需要提取出一个公共方法，这时候需要引用一个新的类，怎么知道这个类具体的名字，目前就是不管什么都放到factory里面。

展开

Geek_3b1096

2021-09-12

周五就遇到只有具体KafkaProducer没有接口没有测试

作者回复: 现学现卖，幸福